Date=26/11/2020
Lecture By=Manish Mahant
Subject ⇒Setting Up Redux

IN PREVIOUS LECTURE **(QUICK RECAP) Date-25/11/2020**	**In Today's Lecture (Overview)**
Redux · An Introduction How Is It Different From MVC And Flux? Benefits Of Redux Functional Programming Where Can Redux Be Used? Building Parts Of Redux	Installation Provider Counter In redux Store In Redux Creating a Store Loading Initial State Dispatching Actions Inside a Redux Store

React Redux is the official React binding for Redux. It lets your React components read data from a Redux store, and dispatch actions to the store to update data.

Installation

React Redux 7.1 requires React 16.8.3 or later.
To use React Redux with your React app:

```
npm install react-redux
```

Provider

React Redux provides <Provider />, which makes the Redux store available to the rest of your app:

```
import React from 'react'
```

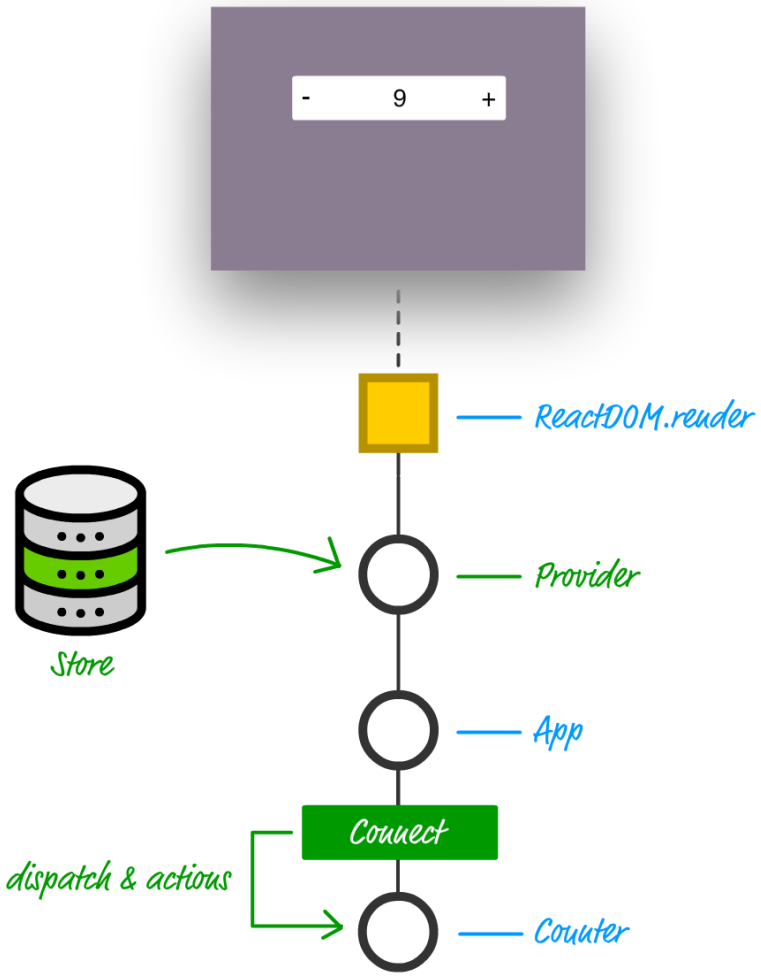
```
import ReactDOM from 'react-dom'

import { Provider } from 'react-redux'
import store from './store'

import App from './App'

const rootElement = document.getElementById('root')
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

Counter In redux



Store In Redux

The Redux store brings together the state, actions, and reducers that make up your app. The store has several responsibilities:

- Holds the current application state inside
- Allows access to the current state via store.getState();
- Allows state to be updated via store.dispatch(action);
- Registers listener callbacks via store.subscribe(listener);
- Handles unregistering of listeners via the unsubscribe function returned by store.subscribe(listener).

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use reducer composition and create multiple reducers that can be combined together, instead of creating separate stores.

Creating a Store#

Every Redux store has a single root reducer function. In the previous section, we created a root reducer function using combineReducers. That root reducer is currently defined in src/reducer.js in our example app. Let's import that root reducer and create our first store.
The Redux core library has a createStore API that will create the store. Add a new file called store.js, and import createStore and the root reducer. Then, call createStore and pass in the root reducer:

src/store.js

```
import { createStore } from 'redux'
import rootReducer from './reducer'

const store = createStore(rootReducer)

export default store
```

Loading Initial State#

createStore can also accept a preloadedState value as its second argument. You could use this to add initial data when the store is created, such as values that were included in an HTML page sent from the server, or persisted in localStorage and read back when the user visits the page again, like this:

storeStatePersistenceExample.js

```
import { createStore } from 'redux'
import rootReducer from './reducer'

let preloadedState
const persistedTodosString = localStorage.getItem('todos')

if (persistedTodosString) {
 preloadedState = {
   todos: JSON.parse(persistedTodosString)
```

```
}
}
```

```
const store = createStore(rootReducer, preloadedState)
```

Dispatching Actions

Now that we have created a store, let's verify our program works! Even without any UI, we can already test the update logic.

src/index.js

Copy

```
// Omit existing React imports

import store from './store'

// Log the initial state
console.log('Initial state: ', store.getState())
// {todos: [....], filters: {status, colors}}

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
const unsubscribe = store.subscribe(() =>
 console.log('State after dispatch: ', store.getState())
)

// Now, dispatch some actions

store.dispatch({ type: 'todos/todoAdded', payload: 'Learn about actions' })
store.dispatch({ type: 'todos/todoAdded', payload: 'Learn about reducers' })
store.dispatch({ type: 'todos/todoAdded', payload: 'Learn about stores' })

store.dispatch({ type: 'todos/todoToggled', payload: 0 })
```

```
store.dispatch({ type: 'todos/todoToggled', payload: 1 })
```

```
store.dispatch({ type: 'filters/statusFilterChanged', payload: 'Active' })
```

```
store.dispatch({
 type: 'filters/colorFilterChanged',
 payload: { color: 'red', changeType: 'selected' }
})
```

```
// Stop listening to state updates
unsubscribe()
// Dispatch one more action to see what happens
store.dispatch({ type: 'todos/todoAdded', payload: 'Try creating a store' })
// Omit existing React rendering logic
```

Inside a Redux Store

It might be helpful to take a peek inside a Redux store to see how it works. Here's a miniature example of a working Redux store, in about 25 lines of code:

miniReduxStoreExample.js

Copy

```
function createStore(reducer, preloadedState) {
 let state = preloadedState
 const listeners = []
```

```
function getState() {
  return state
}
```

```
function subscribe(listener) {
  listeners.push(listener)
  return function unsubscribe() {
    const index = listeners.indexOf(listener)
    listeners.splice(index, 1)
  }
}
```

```
function dispatch(action) {
  state = reducer(state, action)
  listeners.forEach(listener => listener())
}
```

```
dispatch({ type: '@@redux/INIT' })
```

```
return { dispatch, subscribe, getState }
}
```

This small version of a Redux store works well enough that you could use it to replace the actual Redux createStore function you've been using in your app so far. (Try it and see for yourself!) The actual Redux store implementation is longer and a bit more complicated, but most of that is comments, warning messages, and handling some edge cases.

As you can see, the actual logic here is fairly short:

- The store has the current state value and reducer function inside of itself
- getState returns the current state value
- subscribe keeps an array of listener callbacks and returns a function to remove the new callback

- dispatch calls the reducer, saves the state, and runs the listeners
- The store dispatches one action on startup to initialize the reducers with their state
- The store API is an object with {dispatch, subscribe, getState} inside

To emphasize one of those in particular: notice that getState just returns whatever the current state value is. That means that by default, nothing prevents you from accidentally mutating the current state value! This code will run without any errors, but it's incorrect:

```
const state = store.getState()
// ❌ Don't do this - it mutates the current state!
state.filters.status = 'Active'
```

In other words:

- The Redux store doesn't make an extra copy of the state value when you call getState(). It's exactly the same reference that was returned from the root reducer function
- The Redux store doesn't do anything else to prevent accidental mutations. It *is* possible to mutate the state, either inside a reducer or outside the store, and you must always be careful to avoid mutations.

One common cause of accidental mutations is sorting arrays. Calling array.sort() actually mutates the existing array. If we called const sortedTodos = state.todos.sort(), we'd end up mutating the real store state unintentionally.